Gradient Descent and Linear Regression with PyTorch

Edoch

04/08/2022

# Gradient Descent and Linear Regression with PyTorch

Source: [jovian.ai/aakashns/02-linear-regression](https://jovian.ai/aakashns/02-linear-regression)

In this first section a linear regression model is implemented using PyTorch

import numpy as np  
import torch

In the example the objective is to predict the yields for oranges and apples given the values of temperature, rainfall and humidity. Therefore the (linear) models are going to be

## Input data

The input data are the following

# Input (temp, rainfall, humidity)  
inputs = np.array(  
 [[73, 67, 43],  
 [91, 88, 64],  
 [87, 134, 58],  
 [102, 43, 37],  
 [69, 96, 70]],   
 dtype='float32'  
)  
  
inputs

array([[ 73., 67., 43.],  
 [ 91., 88., 64.],  
 [ 87., 134., 58.],  
 [102., 43., 37.],  
 [ 69., 96., 70.]], dtype=float32)

The targets (labels) are the following

# Targets (apples, oranges)  
targets = np.array(  
 [[56, 70],  
 [81, 101],  
 [119, 133],  
 [22, 37],  
 [103, 119]],   
 dtype='float32'  
)  
  
targets

array([[ 56., 70.],  
 [ 81., 101.],  
 [119., 133.],  
 [ 22., 37.],  
 [103., 119.]], dtype=float32)

Transform the input and targets in tensors

inputs = torch.from\_numpy(inputs)  
inputs

tensor([[ 73., 67., 43.],  
 [ 91., 88., 64.],  
 [ 87., 134., 58.],  
 [102., 43., 37.],  
 [ 69., 96., 70.]])

targets = torch.from\_numpy(targets)  
targets

tensor([[ 56., 70.],  
 [ 81., 101.],  
 [119., 133.],  
 [ 22., 37.],  
 [103., 119.]])

## Weights and biases initialization

Weights and biases are initialized with random values following a normal distribution with mean 0 and standard deviation 1. - Weights are collected in a tensor having the where the first dimension (rows) corresponds to the number of models (2, i.e., the number of outputs) and the second dimension (columns) corresponds to the number of features (3, i.e., the number of inputs) - Biases are collected in a tensor having the first dimension equal to the number of models (2)

w = torch.randn((2,3), requires\_grad=True)  
w

tensor([[ 0.7171, -0.4566, 0.3752],  
 [-2.4049, -0.1775, 0.8105]], requires\_grad=True)

b = torch.randn((2), requires\_grad=True)  
b

tensor([ 0.4293, -0.6531], requires\_grad=True)

## The model

The model is

This model can be written in python as a function of x

def regression\_model(x):  
 return x @ w.T + b

where - @ is the matrix multiplication operator - .T (or .t()) outputs the transpose tensor

Moreover, notice that b is a vector, while the result of x @ w.T is a matrix: the summation is computed only after b is broadcasted on the rows to match the matrix shape

## The predictions

The predictions of the model are computed calling the function

preds = regression\_model(inputs)  
preds

tensor([[ 38.3222, -153.2487],  
 [ 49.5216, -183.2430],  
 [ 23.3985, -186.6518],  
 [ 67.8257, -223.5927],  
 [ 32.3434, -126.8929]], grad\_fn=<AddBackward0>)

If the predictions are compared with the targets it can be seen that they are quite different. That’s because the model has not been trained yet, and weights and biases still have random values

## Definition of a Loss function

Improving the model, it must be evaluated how well the model is performing. The model predictions are compared with the actual targets using the following method:

* Calculate the difference between the two matrices (preds and targets).
* Square all elements of the difference matrix to remove negative values.
* Calculate the average of the elements in the resulting matrix.

The result is a single number, known as the **mean squared error** (MSE).

def mse(t1, t2):  
 diff = t1 - t2  
 quad\_diff = diff \* diff  
 mean\_quad\_diff = torch.sum(quad\_diff) / diff.numel()  
 return mean\_quad\_diff

So the loss, in this case, is

mse\_loss = mse(preds, targets)

## Gradient computation

Since w and b have been defined with requires\_grad equal to True, the gradient of the loss function is going to be computed w.r.t w and b

mse\_loss.backward()

The gradients are stored in the .grad attribute of w and b

w.grad, b.grad

(tensor([[ -2534.6836, -4315.5259, -2314.0107],  
 [-22704.0098, -23523.1172, -14637.0957]]),  
 tensor([ -33.9177, -266.7258]))

### Adjust weights and biases to reduce the loss

The loss is a [quadratic function](https://en.wikipedia.org/wiki/Quadratic_function) of weights and biases, and the objective is to find the set of weights where the loss is the lowest. If we plot a graph of the loss w.r.t any individual weight or bias element, it will look like the figure shown below. An important insight from calculus is that the gradient indicates the rate of change of the loss, i.e., the loss function’s [slope](https://en.wikipedia.org/wiki/Slope) w.r.t. the weights and biases.

If a gradient element is **positive**:

* **increasing** the element weight value slightly will **increase** the loss
* **decreasing** the element weight value slightly will **decrease** the loss

|  |
| --- |
| postive-gradient |

If a gradient element is **negative**:

* **increasing** the element weight value slightly will **decrease** the loss
* **decreasing** the element weight value slightly will **increase** the loss

|  |
| --- |
| negative=gradient |

The increase or decrease in the loss by changing a weight element is proportional to the gradient of the loss w.r.t. that element. This observation forms the basis of *the gradient descent* optimization algorithm that we’ll use to improve our model (by *descending* along the *gradient*).

We can subtract from each weight element a small quantity proportional to the derivative of the loss w.r.t. that element to reduce the loss slightly.

Since PyTorch accumulates gradients, to procede the gradients of w and b have to be reset to 0

w.grad.zero\_()  
b.grad.zero\_()  
print(w.grad)  
print(b.grad)

tensor([[0., 0., 0.],  
 [0., 0., 0.]])  
tensor([0., 0.])

Then, we register the current loss to compare it later

old\_mse\_loss = mse(preds, targets)

## Step-by-step: train the model using gradient descent

Follow these steps to train the model 1. Generate predictions 2. Calculate the loss 3. Compute gradients w.r.t the weights and biases 4. Adjust the weights by subtracting a small quantity proportional to the gradient 5. Reset the gradients to zero

### 1. Generate predictions

# Generate predictions  
preds = regression\_model(inputs)  
preds

tensor([[ 38.3222, -153.2487],  
 [ 49.5216, -183.2430],  
 [ 23.3985, -186.6518],  
 [ 67.8257, -223.5927],  
 [ 32.3434, -126.8929]], grad\_fn=<AddBackward0>)

### 2. Calculate the loss (and record the loss for comparison)

# Calculate the loss  
mse\_loss = mse(preds, targets)  
print(mse\_loss)  
old\_mse\_loss = mse\_loss

tensor(37871.8555, grad\_fn=<DivBackward0>)

### 3. Compute gradients w.r.t the weights and biases

# Compute gradients  
mse\_loss.backward()  
print(w.grad)  
print(b.grad)

tensor([[ -2534.6836, -4315.5259, -2314.0107],  
 [-22704.0098, -23523.1172, -14637.0957]])  
tensor([ -33.9177, -266.7258])

### 4 and 5. Adjust the weights by subtracting a small quantity proportional to the gradient and reset the gradients to zero

# Adjust weights & reset gradients  
with torch.no\_grad():  
 w -= w.grad \* 1e-5  
 b -= b.grad \* 1e-5  
 w.grad.zero\_()  
 b.grad.zero\_()

So the weights and biases are updated subtracting from them a value equal to the product between each own gradient and a very small number (10^-5 in this case), to ensure that they don’t get modified by a very large amount. In this way, a small step is taken in the downhill direction of the gradient, not a giant leap. This number is called the **learning rate** of the algorithm.

torch.no\_grad is a context manager used to indicate to PyTorch that it shouldn’t track, calculate, or modify gradients while updating the weights and biases. In this mode, the result of every computation will have requires\_grad=False, even when the inputs have requires\_grad=True.

See https://pytorch.org/docs/stable/generated/torch.no\_grad.html

### Compare the loss between the predictions taken before and after the weights and biases update

new\_preds = regression\_model(inputs)  
new\_mse\_loss = mse(new\_preds, targets)  
print(f'old loss: {old\_mse\_loss}')  
print(f'new loss: {new\_mse\_loss}')

old loss: 37871.85546875  
new loss: 25911.462890625

The loss reduced by a lot

## Train for multiple epochs

To reduce the loss further, repeat the process of adjusting the weights and biases using the gradients multiple times. Each iteration is called an **epoch**.

Train the model for 1000 epochs.

# reset w and b  
from tqdm.notebook import tqdm  
  
  
w = torch.randn((2,3), requires\_grad=True)  
b = torch.randn((2), requires\_grad=True)  
  
def regression\_model(inputs,w,b):  
 return inputs@w.T+b  
  
def mse(t1,t2):  
 diff = t1-t2  
 return torch.sum(diff\*diff)/diff.numel()  
  
loss\_history = []  
  
for i in tqdm(range(1000)):  
  
 # generate predictions  
 preds = regression\_model(inputs,w,b)  
   
 # calculate the loss  
 mse\_loss = mse(preds,targets)  
 loss\_history.append(mse\_loss)  
  
 # compute the gradients  
 mse\_loss.backward()  
  
 # update w and b  
 with torch.no\_grad():  
 w -= w.grad\*1e-5  
 b -= b.grad\*1e-5  
 w.grad.zero\_()  
 b.grad.zero\_()  
  
print(f'starting loss\t{loss\_history[0]}')  
print(f'ending loss\t{loss\_history[-1]}')

0%| | 0/1000 [00:00<?, ?it/s]

starting loss 31568.06640625  
ending loss 54.96778106689453

# Linear regression using PyTorch built-ins

PyTorch provides a built-in for linear regression models.

Import the torch.nn package, which contains utility classes for building neural networks

import torch.nn as nn

Import the inputs and targets

# Input (temp, rainfall, humidity)  
inputs = np.array(  
 [[73, 67, 43],  
 [91, 88, 64],  
 [87, 134, 58],  
 [102, 43, 37],  
 [69, 96, 70],  
 [74, 66, 43],  
 [91, 87, 65],  
 [88, 134, 59],  
 [101, 44, 37],  
 [68, 96, 71],  
 [73, 66, 44],  
 [92, 87, 64],  
 [87, 135, 57],  
 [103, 43, 36],  
 [68, 97, 70]],   
 dtype='float32'  
)  
  
# Targets (apples, oranges)  
targets = np.array(  
 [[56, 70],  
 [81, 101],  
 [119, 133],  
 [22, 37],  
 [103, 119],  
 [57, 69],  
 [80, 102],  
 [118, 132],  
 [21, 38],  
 [104, 118],  
 [57, 69],  
 [82, 100],  
 [118, 134],  
 [20, 38],  
 [102, 120]],  
 dtype='float32'  
)  
  
inputs = torch.from\_numpy(inputs)  
targets = torch.from\_numpy(targets)

## Dataset and DataLoader

### TensorDataset

TensorDataset allows access to rows from inputs and targets as tuples, and provides standard APIs for working with many different types of datasets in PyTorch.

See https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

It allows to access a small section of the training data using the array indexing notation ([0:3] in the above code). It returns a tuple with two elements: - the first element contains the input variables for the selected rows - the second contains the targets

from torch.utils.data import TensorDataset

# Define the TensorDataset  
train\_ds = TensorDataset(inputs, targets)  
train\_ds[0:3]

(tensor([[ 73., 67., 43.],  
 [ 91., 88., 64.],  
 [ 87., 134., 58.]]),  
 tensor([[ 56., 70.],  
 [ 81., 101.],  
 [119., 133.]]))

### DataLoader

A DataLoader is a Python iterable over a dataset. It can split the data into batches of a predefined size while training. It also provides other utilities like shuffling and random sampling of the data.

from torch.utils.data import DataLoader

# Define data loader  
train\_dl = DataLoader(train\_ds, batch\_size = 5, shuffle=True)

The batch\_size parameter divides the dataset into batches: during each epoch, weights and biases are updated computing the gradients over each batch, so speeding up the gradient descent since more than one update (in this case 3, since the samples are 15 divided into batches of 5 samples) occurs during each epoch. Iterating over the dataloader, it returns one batch of data with the given batch size.

The shuffle parameter set to True orders to the dataloader to shuffle the training data before creating batches. Shuffling helps randomize the input to the optimization algorithm, leading to a faster reduction in the loss.

# Example: the first batch of the dataset  
list(train\_dl)[0]

[tensor([[ 73., 67., 43.],  
 [103., 43., 36.],  
 [101., 44., 37.],  
 [ 69., 96., 70.],  
 [ 73., 66., 44.]]),  
 tensor([[ 56., 70.],  
 [ 20., 38.],  
 [ 21., 38.],  
 [103., 119.],  
 [ 57., 69.]])]

### nn.Linear

The nn.Linear class creates a linear model (), and automatically initializes weights and biases

# Define model  
linear\_model = nn.Linear(3, 2)  
print(linear\_model.weight)  
print(linear\_model.bias)

Parameter containing:  
tensor([[ 0.2004, -0.1433, 0.1634],  
 [-0.4712, -0.5417, -0.2353]], requires\_grad=True)  
Parameter containing:  
tensor([-0.3474, 0.0779], requires\_grad=True)

To see the list of all the weights and bias matrices present in the model use the .parameters method

# Parameters  
list(linear\_model.parameters())

[Parameter containing:  
 tensor([[ 0.2004, -0.1433, 0.1634],  
 [-0.4712, -0.5417, -0.2353]], requires\_grad=True),  
 Parameter containing:  
 tensor([-0.3474, 0.0779], requires\_grad=True)]

## Loss Function

Also the mse loss function is already available in PyTorch as a built-in loss function, and it is called mse\_loss

Built-in loss functions and other utlities are collected in the nn.functional module

# Import nn.functional  
from torch.nn.functional import mse\_loss

loss = mse\_loss(linear\_model(inputs), targets)  
print(loss)

tensor(22213.8789, grad\_fn=<MseLossBackward0>)

## Optimizer

Instead of manually manipulating the model’s weights & biases using gradients, the optimizer optim.SGD can be used. SGD is short for “stochastic gradient descent”. The term *stochastic* indicates that samples are selected in random batches instead of as a single group.

Optimizers are collected in the optim module

# Define optimizer  
opt\_SDG = torch.optim.SGD(linear\_model.parameters(), lr=1e-5)

I’m passing as the first argument of the SGD class a list containing the model parameters

## Model training

Steps to implement gradient descent:

1. Generate predictions
2. Calculate the loss
3. Compute gradients w.r.t the weights and biases
4. Adjust the weights by subtracting a small quantity proportional to the gradient
5. Reset the gradients to zero

The only change is that now the dataset is divided into batches, instead of processing the entire training data in every iteration.

It is useful to define a utility function fit that trains the model for a given number of epochs.

# Utility function to train the model  
def fit(num\_epochs, model, loss\_fn, opt, train\_dl):  
   
 # Repeat for given number of epochs  
 for epoch in range(num\_epochs):  
   
 # Train with batches of data  
 for xb,yb in train\_dl:  
   
 # 1. Generate predictions  
 pred = model(xb)  
   
 # 2. Calculate loss  
 loss = loss\_fn(pred, yb)  
   
 # 3. Compute gradients  
 loss.backward()  
   
 # 4. Update parameters using gradients  
 opt.step()  
   
 # 5. Reset the gradients to zero  
 opt.zero\_grad()  
   
 # Print the progress  
 if (epoch+1) % 10 == 0:  
 print(f'Epoch [{epoch+1}/{num\_epochs}], Loss: {loss.item():.4f}')

Notice that: - dataloader is used to get batches of data for every iteration - parameters (weights and biases) are update not manually but automatically, using - opt.step to perform the update - opt.zero\_grad to reset the gradients to zero. - a log statement that prints the loss from the last batch of data for every 10th epoch to track training progress has been added (the loss.item() method returns the actual value stored in the loss tensor)

Train the model for 100 epochs

fit(  
 num\_epochs=100,  
 model=linear\_model,  
 loss\_fn=mse\_loss,  
 opt=opt\_SDG,  
 train\_dl=train\_dl  
)

Epoch [10/100], Loss: 744.7711  
Epoch [20/100], Loss: 107.7794  
Epoch [30/100], Loss: 71.2221  
Epoch [40/100], Loss: 73.8659  
Epoch [50/100], Loss: 59.5663  
Epoch [60/100], Loss: 30.8944  
Epoch [70/100], Loss: 51.4250  
Epoch [80/100], Loss: 49.9322  
Epoch [90/100], Loss: 27.3745  
Epoch [100/100], Loss: 12.0347